ing trends. Implement best

A Lightweight, Safe, Portable, and High-Performance Runtime for Dapr

Key Takeaways

- Dapr is a versatile framework for building microservices.
- WebAssembly VMs, such as WasmEdge, provide high-performance and secure runtimes for microservice applications.
- WebAssembly-based microservices can be written in a number of programming languages, including Rust, C/C++, Swift, and JavaScript.
- WebAssembly programs are embedded into Dapr sidecar applications, and hence can be portable and agnostic to the Dapr host environment.
- The WasmEdge SDK provides an easy way to build microservices for Tensorflow inference.

Since its release in 2019, Dapr (Distributed Application Runtime) has quickly become a very popular open-source framework for building microservices. It provides building blocks and pre-packaged services that are commonly used in distributed applications, such as service invocation, state management, message queues, resource bindings and triggers, mTLS secure connections, and service monitoring. Distributed application developers can utilize and consume web-based APIs exposed by those building blocks at runtime. These applications are commonly known as microservices and run as sidecars. Dapr is an example of the Multi-Runtime Microservices Architecture, as described by InfoQ author Bilgin Ibryam.

> *Dapr's sidecar pattern is very much like a service mesh. However, unlike traditional service mesh which aims to manage applications without any code change, Dapr applications need to integrate and actively utilize external Dapr building block services.*

The microservice applications in Dapr sidecars could be native client (NaCl) applications compiled from languages like Go and Rust, or managed language applications written in Python or JavaScript. In other words, the sidecar applications could have their own language runtimes. The sidecar model allows Dapr to support “any language, any framework, anywhere” for its applications.

WebAssembly and WasmEdge

Dapr can run sidecar applications directly on the OS or through an application container like Docker. The container offers benefits such as portability, ease of deployment, and security, but it also brings significant overheads.

In this article, we present a new approach to run Dapr sidecar applications. We use a simple NaCl written in Rust or Go to listen for API requests to the microservice. It passes the request data to a WebAssembly runtime for processing. The business logic of the microservice is a WebAssembly function created and deployed by an application developer.

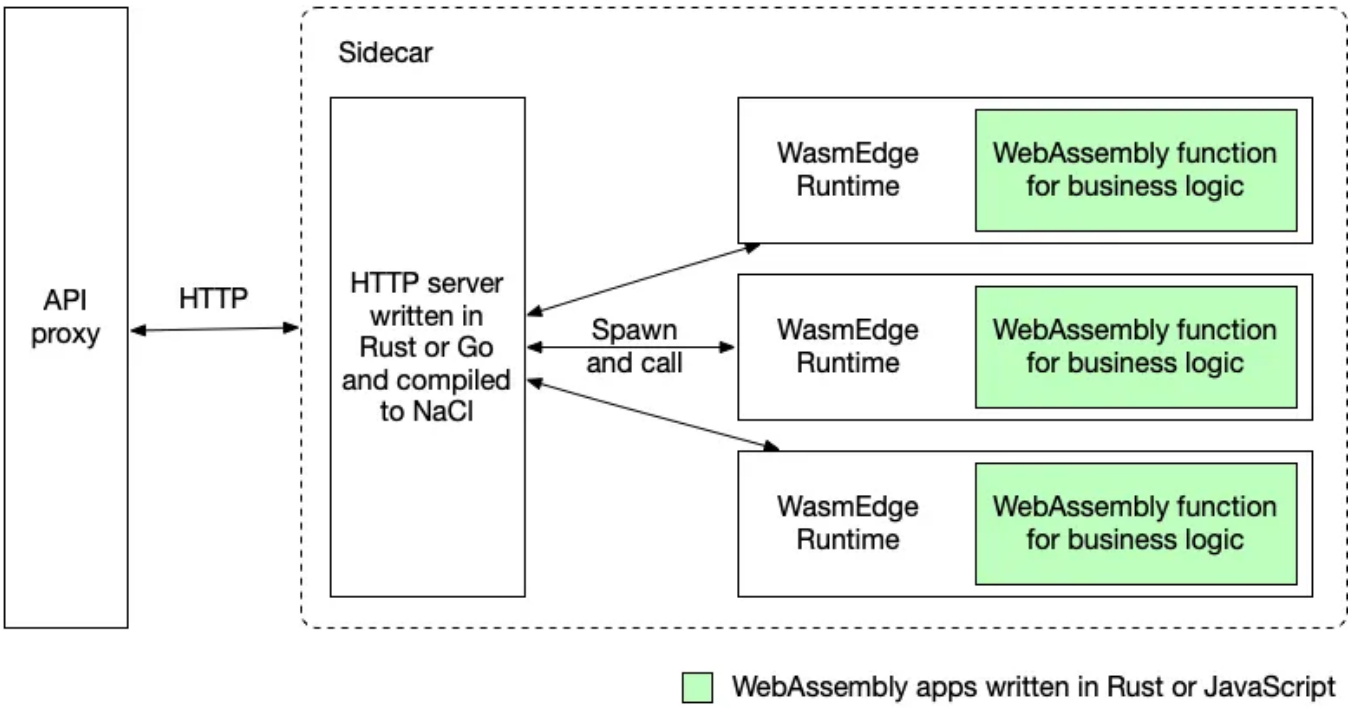

Figure 1. A Dapr microservice with a WebAssembly function.

The WebAssembly runtime is well suited to execute the business logic function.

- WebAssembly programs could run as fast as compiled machine-native binaries and consume much less resources than containers.

- WebAssembly supports high-performance languages like C/C++, Rust, Swift, and Kotlin. It could also support high-level languages like JavaScript and DSLs (Domain Specific Languages).
- WebAssembly programs are portable and can be easily deployed across different operating systems and hardware platforms.
- WebAssembly provides a secure sandbox that isolates applications at runtime. Developers can limit the program's access to OS or other resources by declaring a security policy.

The table below summarizes the pros and cons of different approaches for the sidecar application.

	NaCl	Application Runtimes (eg Node & Python)	Docker-like Container	WebAssembly
Performance	Great	Poor	OK	Great
Resource footprint	Great	Poor	Poor	Great
Isolation	Poor	OK	OK	Great
Safety	Poor	OK	OK	Great
Portability	Poor	Great	OK	Great
Security	Poor	OK	OK	Great
Language and framework choice	N / A	N / A	Great	OK
Ease of use	OK	Great	Great	OK
Manageability	Poor	Poor	Great	Great

WasmEdge is a leading cloud-native WebAssembly runtime hosted by the CNCF (Cloud Native Computing Foundation) / Linux Foundation. It is the fastest WebAssembly runtime in the market today. WasmEdge supports all standard WebAssembly extensions as well as proprietary extensions for Tensorflow inference, KV store, and image processing, etc. Its compiler toolchain supports not only WebAssembly languages such as C/C++, Rust, Swift, Kotlin, and AssemblyScript but also regular JavaScript.

A WasmEdge application can be embedded into a C program, a Go program, a Rust program, a JavaScript program, or the operating system's CLI. The runtime can be managed by Docker tools (eg CRI-O), orchestration tools (eg K8s), serverless platforms (eg Vercel, Netlify, AWS Lambda, Tencent SCF), and data streaming frameworks (eg YoMo and Zenoh).

In this article, I will demonstrate how to use WasmEdge as a sidecar application runtime for Dapr.

Quick start

First you need to install Go, Rust, Dapr, WasmEdge, and the rustwasmc compiler tool.

Next, fork or clone the demo application from Github. You can use this repo as your own application template.

```
$ git clone https://github.com/second-state/dapr-wasm
```

The demo has 3 Dapr sidecar applications.

- The web-port project provides a public web service for a static HTML page. This is the application's UI.
- The image-api-rs project provides a WasmEdge microservice to turn an input image into a grayscale image using the grayscale function. It demonstrates the use of Rust SDKs for Dapr and WasmEdge.
- The image-api-go project provides a WasmEdge microservice to recognize and classify the object on an input image using the classify function. It demonstrates the use of Go SDKs for Dapr and WasmEdge.

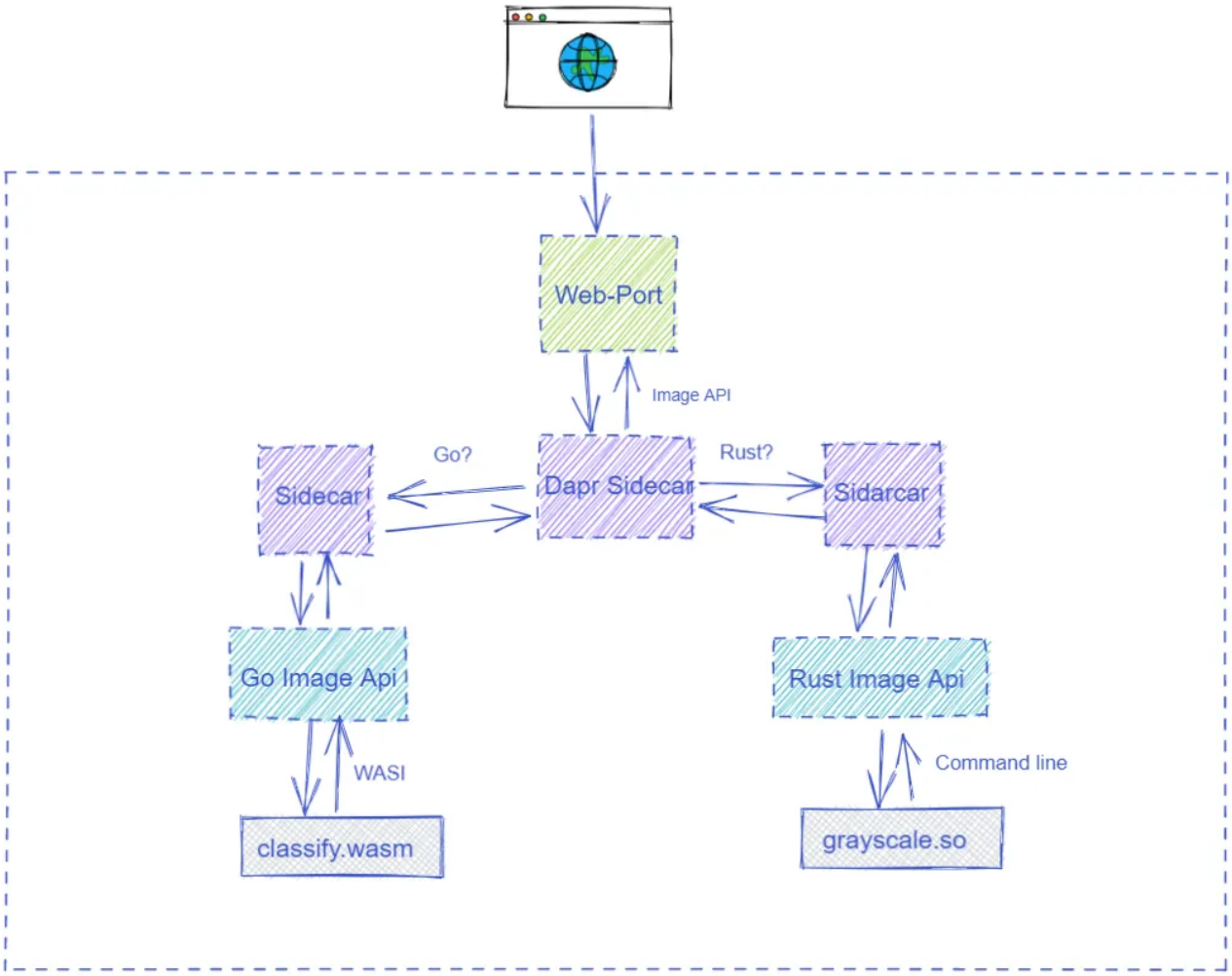

Figure 2. Dapr sidecar microservices in the demo application.

You can follow the instructions in the README to start the sidecar services. Here are commands to build the WebAssembly functions and start the 3 sidecar services.

```
# Build the classify and grayscale WebAssembly functions, and
$ cd functions/grayscale
$ ./build.sh
$ cd ../../
$ cd functions/classify
$ ./build.sh
$ cd ../../

# Build and start the web service for the application UI
$ cd web-port
$ go build
```

```
$ ./run_web.sh
$ cd ../

# Build and start the microservice for image processing (gray
$ cd image-api-rs
$ cargo build
$ ./run_api_rs.sh
$ cd ../

# Build and start the microservice for tensorflow-based image
$ cd image-api-go
$ go build --tags "tensorflow image"
$ ./run_api_go.sh
$ cd ../
```

Finally, you should be able to see the web UI in your browser.

Figure 3. The demo application in action.

The two WebAssembly functions

We have two functions written in Rust and compiled into WebAssembly. They are deployed in the sidecar microservices to perform the actual work of image processing and classification.

While our example WebAssembly functions are written in Rust, you can compile functions written in C/C++, Swift, Kotlin, and AssemblyScript to WebAssembly. WasmEdge also provides support for functions written in JavaScript and DSLs.

The grayscale function is a Rust program that reads image data from `STDIN` and writes the grayscale image into `STDOUT`.

```
use image::{ImageFormat, ImageOutputFormat};
use std::io::{self, Read, Write};

fn main() {
    let mut buf = Vec::new();
    io::stdin().read_to_end(&mut buf).unwrap();

    let image_format_detected: ImageFormat = image::guess_form
    let img = image::load_from_memory(&buf).unwrap();
    let filtered = img.grayscale();
    let mut buf = vec![];
    match image_format_detected {
        ImageFormat::Gif => {
            filtered.write_to(&mut buf, ImageOutputFormat::Gif
        }
        _ => {
            filtered.write_to(&mut buf, ImageOutputFormat::Png
        }
    };
    io::stdout().write_all(&buf).unwrap();
    io::stdout().flush().unwrap();
}
```

We use rustwasmc to build it and then copy it to the image-api-rs sidecar.

```
$ cd functions/grayscale
$ rustup override set 1.50.0
$ rustwasmc  build --enable-ext
$ cp ./pkg/grayscale.wasm ../../image-api-rs/lib
```

The classify function is a Rust function that takes a byte array for image data as input and returns a string for the classification. It uses the WasmEdge TensorFlow API.

```
use wasmedge_tensorflow_interface;

pub fn infer_internal(image_data: &[u8]) -> String {
    let model_data: &[u8] = include_bytes!("models/mobilenet_v
    let labels = include_str!("models/mobilenet_v1_1.0_224/lab

    let flat_img = wasmedge_tensorflow_interface::load_jpg_ima

    let mut session = wasmedge_tensorflow_interface::Session::
        &model_data,
        wasmedge_tensorflow_interface::ModelType::TensorFlowLi
    );
    session
        .add_input("input", &flat_img, &[1, 224, 224, 3])
        .run();
    let res_vec: Vec<u8> = session.get_output("MobilenetV1/Pre

    // ... Map the probabilities in res_vec to text labels in

    if max_value > 50 {
        format!(
            "It {} a <a href='https://www.google.com/search?q=
            confidence.to_string(),
            class_name,
            class_name
```

```
        )
    } else {
        format!("It does not appears to be any food item in th
    }
}
```

We use rustwasmc to build it and then copy it to the image-api-go sidecar.

```
$ cd functions/classify
$ rustup override set 1.50.0
$ rustwasmc  build --enable-ext
$ cp ./pkg/classify_bg.wasm ../../image-api-go/lib/classify_b
```

In the next three sections, we will look into those three sidecar services.

The image processing sidecar

The image-api-rs sidecar application is written in Rust. It should already have the WebAssembly function lib/grayscale.wasm installed from the previous step. Please refer to the functions/bin/install.sh script to install the WasmEdge Runtime binary `lib/wasmedge-tensorflow-lite` and its dependencies.

The sidecar microservice runs a Tokio-based event loop that listens for incoming HTTP requests at the path `/api/image`.

```
#[tokio::main]
pub async fn run_server(port: u16) {
    pretty_env_logger::init();

    let home = warp::get().map(warp::reply);

    let image = warp::post()
        .and(warp::path("api"))
        .and(warp::path("image"))
        .and(warp::body::bytes())
        .map(|bytes: bytes::Bytes| {
```

```
            let v: Vec<u8> = bytes.iter().map(|&x| x).collect(
            let res = image_process(&v);
            Ok(Box::new(res))
        });

    let routes = home.or(image);
    let routes = routes.with(warp::cors().allow_any_origin());

    let log = warp::log("dapr_wasm");
    let routes = routes.with(log);
    warp::serve(routes).run((Ipv4Addr::UNSPECIFIED, port)).awa
}
```

Once it receives an image file in the HTTP POST request, it invokes a WebAssembly function in WasmEdge to perform the image processing task. It creates a WasmEdge instance to interact with the WebAssembly program.

```
pub fn image_process(buf: &Vec<u8>) -> Vec<u8> {
    let mut child = Command::new("./lib/wasmedge-tensorflow-li
        .arg("./lib/grayscale.wasm")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("failed to execute child");
    {
        // limited borrow of stdin
        let stdin = child.stdin.as_mut().expect("failed to get
        stdin.write_all(buf).expect("failed to write to stdin"
    }
    let output = child.wait_with_output().expect("failed to wa
    output.stdout
}
```

The following Dapr CLI command starts the microservice in the Dapr runtime environment.

```
$ cd image-api-rs
$ sudo dapr run --app-id image-api-rs \
        --app-protocol http \
        --app-port 9004 \
        --dapr-http-port 3502 \
        --components-path ../config \
        --log-level debug \
        ./target/debug/image-api-rs
$ cd ../
```

The Tensorflow sidecar

The image-api-go sidecar application is written in Go. It should already have the WebAssembly function `lib/classify_bg.wasm` installed from the previous step. Please refer to the functions/bin/install.sh script to install the WasmEdge Runtime Go SDK.

The sidecar microservice runs an event loop that listens for incoming HTTP requests at the path `/api/image`.

```
func main() {
   s := daprd.NewService(":9003")

   if err := s.AddServiceInvocationHandler("/api/image", imag
       log.Fatalf("error adding invocation handler: %v", err)
   }

   if err := s.Start(); err != nil && err != http.ErrServerCl
       log.Fatalf("error listenning: %v", err)
   }
}
```

Once it receives an image file in the HTTP POST request, it invokes a WebAssembly function in WasmEdge to perform the Tensorflow-based image classification task. It utilizes the Go API for WasmEdge to interact with the WebAssembly program.

```
func imageHandlerWASI(_ context.Context, in *common.Invocatio
    image := in.Data

    var conf = wasmedge.NewConfigure(wasmedge.REFERENCE_TYPES)
    conf.AddConfig(wasmedge.WASI)
    var vm = wasmedge.NewVMWithConfig(conf)

    var wasi = vm.GetImportObject(wasmedge.WASI)
    wasi.InitWasi(
        os.Args[1:],     /// The args
        os.Environ(),    /// The envs
        []string{".:."}, /// The mapping directories
        []string{},      /// The preopens will be empty
    )

    /// Register WasmEdge-tensorflow and WasmEdge-image
    var tfobj = wasmedge.NewTensorflowImportObject()
    var tfliteobj = wasmedge.NewTensorflowLiteImportObject()
    vm.RegisterImport(tfobj)
    vm.RegisterImport(tfliteobj)
    var imgobj = wasmedge.NewImageImportObject()
    vm.RegisterImport(imgobj)

    vm.LoadWasmFile("./lib/classify_bg.wasm")
    vm.Validate()
    vm.Instantiate()

    res, err := vm.ExecuteBindgen("infer", wasmedge.Bindgen_re
    ans := string(res.([]byte))

    vm.Delete()
    conf.Delete()
```

```
    out = &common.Content{
        Data:        []byte(ans),
        ContentType: in.ContentType,
        DataTypeURL: in.DataTypeURL,
    }
    return out, nil
}
```

The following Dapr CLI command starts the microservice in the Dapr runtime environment.

```
$ cd image-api-go
$ sudo dapr run --app-id image-api-go \
        --app-protocol http \
        --app-port 9003 \
        --dapr-http-port 3501 \
        --log-level debug \
        --components-path ../config \
        ./image-api-go
$ cd ../
```

The web UI sidecar

The web UI service web-port is a simple web server written in Go. It serves static HTML and JavaScript files from the static folder and sends images uploaded to /api/hello to the grayscale or classify sidecars' /api/image endpoints.

```
func main() {
    http.HandleFunc("/static/", staticHandler)
    http.HandleFunc("/api/hello", imageHandler)
    println("listen to 8080 ...")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

```
func staticHandler(w http.ResponseWriter, r *http.Request) {
    // ... read and return the contents of HTML CSS and JS fil
}

func imageHandler(w http.ResponseWriter, r *http.Request) {
    // ... ...
    api := r.Header.Get("api")
    if api == "go" {
        daprClientSend(body, w)
    } else {
        httpClientSend(body, w)
    }
}

// Send to the image-api-go sidecar (classify) via the Dapr A
func daprClientSend(image []byte, w http.ResponseWriter) {
    // ... ...
    resp, err := client.InvokeMethodWithContent(ctx, "image-ap
    // ... ...
}

// Send to the image-api-rs sidecar (grayscale) via the HTTP
func httpClientSend(image []byte, w http.ResponseWriter) {
    // ... ...
    req, err := http.NewRequest("POST", "http://localhost:3502
    // ... ...
}
```

The JavaScript in page.js simply uploads images to the web-port sidecar's `/api/hello` endpoint and the web-port will request the classify or grayscale microservice based on the request header `api`.

```
function runWasm(e) {
    const reader = new FileReader();
    reader.onload = function (e) {
        setLoading(true);
```

```
        var req = new XMLHttpRequest();
        req.open("POST", '/api/hello', true);
        req.setRequestHeader('api', getApi());
        req.onload = function () {
            // ...  display results ...
        };
        const blob = new Blob([e.target.result], {
            type: 'application/octet-stream'
        });
        req.send(blob);
    };
    console.log(image.file)
    reader.readAsArrayBuffer(image.file);
}
```

The following Dapr CLI command starts the web service for the static UI files.

```
$ cd web-port
$ sudo dapr run --app-id go-web-port \
        --app-protocol http \
        --app-port 8080 \
        --dapr-http-port 3500 \
        --components-path ../config \
        --log-level debug \
        ./web-port
$ cd ../
```

That's it. You now have a three part distributed application written in two languages!

What's next

As we have demonstrated, there is a lot of synergy between Dapr's distributed network runtime and WasmEdge's universal language runtime. This approach can be generalized and applied to other service mesh or distributed application frameworks. Unlike Dapr, many service meshes can only operate in Kubernetes as their control plane and hence are dependent on the Kubernetes API. WasmEdge is a Kubernetes compatible runtime and could play an important role as a lightweight container alternative to run microservices. Stay tuned!

About the Author

Dr. Michael Yuan is the author of five books on software engineering. His latest book,Building Blockchain Apps, was published by Addison-Wesley in Dec 2019. Dr. Yuan is the co-founder of Second State, a startup that brings WebAssembly and Rust technologies to cloud, blockchain, and AI applications. Second State enables developers to deploy fast, safe, portable, and serverless Rust functions on Node.js. Stay in touch by subscribing to the WebAssembly.Today newsletter.

Discuss